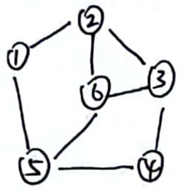
第六章 图

一、基本概念

1. 定义：图G由顶点集V和边集E组成，G=(V,E)。



V={1,2,3,4,5,6}

E={(1,2),(2,3),(3,4),(4,5),(1,5),(5,6),(2,6),(3,6)}

① 顶点集V一定非空，边集E可以为空。
② 空表，空树，无空图。
③ |V| 顶点个数，阶；
|E| 边个数。

2. 常见术语

(1) 无向图：全部由无向边构成的图。

(2) 有向图：全部由有向边构成的图。

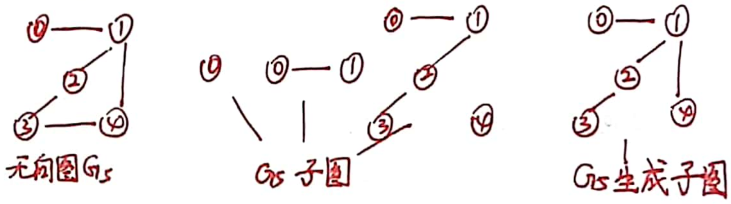
(3) ① 简单图：不存在重复边，不存在顶点到自身的边。

② 多重图：某两个顶点之间的边数大于1条，又允许顶点通过一条边和自身相连。

(4) 完全图（简单完全图）：
- 边数 $\frac{n(n-1)}{2}$：完全无向图
- 边数 $n(n-1)$：完全有向图

(5) ① G的子图：所有顶点和边都属于图G的图。

② G的生成子图：含有G的所有顶点的子图。



(6) ① V和W连通：无向图中，V到W的路径存在。

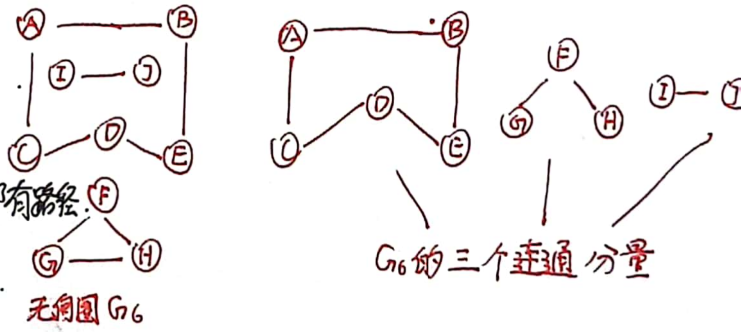
[无向图] ② 连通图：图中任意、两个顶点都是连通的。

③ 连通分量：无向图中的极大连通子图。

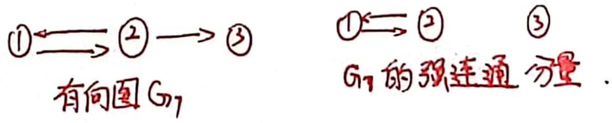
(7) ① V和W强连通：有向图中，从V到W和从W到V都有路径。

[有向图] ② 强连通图：图中任何一对顶点都是强连通的。

③ 强连通分量：有向图中的极大连通子图。

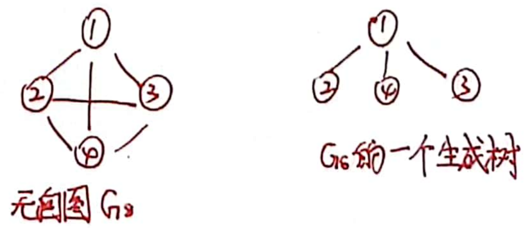


(8) ① 生成树：包含图中全部顶点的一个极小连通图。



② 生成森林：非连通图中，连通分量的生成树构成了非连通图的生成森林。

[注]
- 极大连通子图：子图连通且包含尽可能多的顶点和边
- 极小连通子图：子图连通又要使边数最少。



(9) ① 顶点的度：图中与该顶点相关联边的数目。

② 入度：指向该顶点的边的数目。

③ 出度：从该顶点出去的边的数目。

- 顶点n，边e的有向图：出度和 = 入度和 = e
- 顶点n，边e的无向图：度和 = 2e

(10) ①边权：边上的数值.

②边网：边上标识权的图.

(11) ①稠密图：边多.

②稀疏图：边少. $|E|<|V|\log|V|$

(12) ①路径：在一个图中，路径是从顶点u到顶点v所经过的顶点序列.

②路径长度：该路径上边的数目.

③回路：第一个顶点和最后一个顶点相同的路径.

(13) ①简单路径：顶点不重复出现的路径.

②简单回路：除第一个和最后一个顶点，其余顶点不重复出现的回路.

(14) 距离：从u到v的最短路径长度.

(15) 有向树：一个顶点的入度为0，其余顶点的入度均为1的有向图.

3. 常考结论：

(1) 无向图边数 ×2 = 各顶点度数之和

有向图边数 = 各顶点入度之和 = 各顶点出度之和.

(2) 一个连通图的生成树是一个极小连通子图，是无环的.

(3) 完全无向图：边数 $\frac{n(n-1)}{2}$

完全有向图：边数 $n(n-1)$

(4) 对于一个有n个顶点的图：

①若是连通无向图，边的个数至少为 $n-1$（一棵树）

非连通 ~ ~ 至多为 C_{n-1}^{2}

②若是强连通有向图，边的个数至少为 n（回路）

一、图的存储

1. 邻接矩阵法：

带权图

$$A[i][j]=\begin{cases}1,\ w_{ij} & \langle v_i,v_j\rangle 是边\\ 0,\ 0或\infty & \langle v_i,v_j\rangle 不是边\end{cases}$$

① 用两个数组表示图．

② 一维数组存储图中顶点信息．

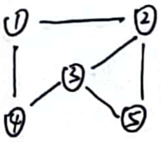
③ 二维数组存储图中的边或弧的信息．

```
#define MaxVertexNum 100
typedef char VertexType;   //顶点对应的数据类型
typedef int EdgeType;      //边对应的数据类型
typedef struct{
    VertexType vex[MaxVertexNum];                 //顶点表
    EdgeType edge[MaxVertexNum][MaxVertexNum];    //邻接矩阵，边表
    int vexnum, arcnum;                           //当前顶点数和边数
}MGraph;
```

① 空间复杂度 $O(|V|^2)$

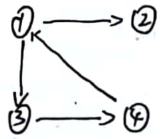
② 适合稠密图

③ A^n 的元素 $A^n[i][j]$ 等于由顶点 i 到顶点 j 的长度为 n 的路径的数目



$$A_1=\begin{bmatrix}0&1&0&1&0\\1&0&1&0&1\\0&1&0&1&1\\1&0&1&0&0\\0&1&1&0&0\end{bmatrix}$$

无向图 G_1 及其邻接矩阵



$$A_2=\begin{bmatrix}0&1&1&0\\0&0&0&0\\0&0&0&1\\1&0&0&0\end{bmatrix}$$

有向图 G_2 及其邻接矩阵

① 无向图的邻接矩阵一定是一个对称矩阵(唯一)，可压缩存储．

② 顶点 i 的度：第 i 行(或第 i 列)非零元素的个数．

① 有向图顶点 i 的出度：第 i 行非零元素的个数

入度：第 i 列非零元素的个数

2. 邻接表法：

data	firstarc

顶点域　边表头指针

顶点表结点

adjvex	nextarc

邻接点域　指针域

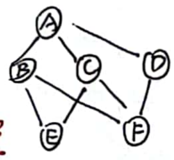
边表结点

对于有向图则称为出边表

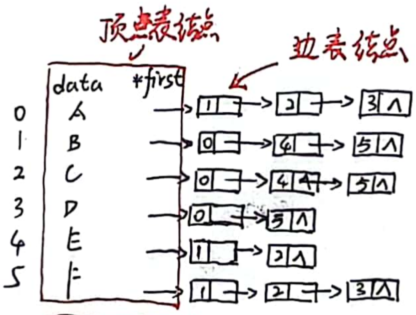
① 顶点用一个一维数组存储．

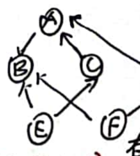
② 每个顶点的所有邻接点构成一个线性表，用单链表存储．

```
#define MaxVertexNum 100
typedef struct ArcNode{          //边表结点
    int adjvex;                  //该弧所指向的顶点的位置
    struct ArcNode *nextarc;     //指向下一条弧的指针
    //InfoType info;             //网的边权值
}ArcNode;
typedef struct VNode{            //顶点表结点
    VertxType data;              //顶点信息
    ArcNode *firstarc;           //指向第一条依附该顶点的弧的指针
}VNode, AdjList[MaxVertexNum];
typedef struct{
    AdjList vertices;            //邻接表
    int vexnum, arcnum;          //图的顶点数和弧数
}ALGraph;
```

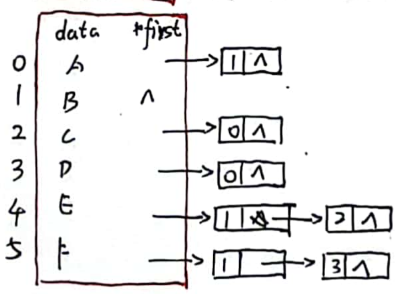


无向图





有向图



① 无向图存储空间 $O(|V|+2|E|)$．

有向图存储空间 $O(|V|+|E|)$．

② 适于稀疏图

③ 无向图顶点 i 度：对应邻接表中的边表结点个数

有向图顶点 i 出度：对应邻接表中的边表结点个数

④ 图的邻接表表示不唯一．

3. 十字链表（针对有向图）：

① 把邻接表和逆邻接表整合在一起．

② 既容易找到以vi为尾的弧，也容易找到以vi为头的弧．

③ 容易求得顶点的出度和入度．

④ 创建图算法的时间复杂度和邻接表相同．

⑤ 空间复杂度 $O(|V|+|E|)$

⑥ 图的十字链表不是唯一的，但一个十字链表唯一确定一个图．

弧尾结点　弧头结点　权值

tailvex	headvex	hlink	tlink	(info)

弧结点

弧头相同的下一条弧　弧尾相同的下一条弧

数据域

data	firstin	firstout

顶点结点

该顶点作为弧头的第一条弧　该顶点作为弧尾的第一条弧

4. 邻接多重表（针对无向图）：

① 仿照十字链表的方式，对边表结点的结构进行一些构造．

② 同一条边在邻接表中用两个结点表示，而在邻接多重表中只有一个结点．

③ 空间复杂度 $O(|V|+|E|)$

边结点、

ivex	ilink	jvex	jlink	(info)

权值

边的两个顶点编号　指向下一条依附于顶点j的边　指向下一条依附于顶点i的边

顶点结点、

data	firstedge

数据域　指向第一条依附于该顶点的边．

5. 总结与对比

	邻接矩阵	邻接表	十字链表	邻接多重表
空间复杂度	$O(\|V\|^2)$	无向图：$O(\|V\|+2\|E\|)$ 有向图：$O(\|V\|+\|E\|)$	$O(\|V\|+\|E\|)$	$O(\|V\|+\|E\|)$
找相邻边	遍历对应行或列的时间复杂度 $O(\|V\|)$	找有向图的入度必须遍历整个邻接表	很方便	很方便
删除边或顶点	删除边方便，删除点，需要大量移动数据	无向图中删除边或顶点，都不方便	很方便	很方便
适用于	稠密图	稀疏图和其他	只能存有向图	只能无向图
表示方式	唯一	不唯一	不唯一	不唯一

① 有向图邻接表中，删除某个顶点的所有边的时间复杂度 $O(n+e)$．

② 建立无向图邻接表的时间复杂度 $O(n+e)$．

三、图的遍历

1. 广度优先搜索 BFS

① 相当于树的层次遍历。

② 借助一个辅助队列。 空间复杂度 $O(|V|)$

(1) BFS 算法伪代码：

```
bool  visited[MAX_VERTEX_NUM];       // 访问标记数组
void  BFSTraverse (Graph G) {
    for (int i=0 ; i<G.vexnum ; i++)
        visited[i] = FALSE ;
    InitQueue (Q);                   // 初始化辅助队列 Q
    for (int i=0 ; i<G.vexnum ; i++)  // 从0号顶点遍历
        if (!visited[i])             // 对每个连通分量调用一次BFS()
            BFS (G, i);              // 若vi未访问过，从vi开始调用BFS.
}
```

(2) 邻接表实现 BFS： 时间复杂度 $O(|V|+|E|)$

```
void  BFS (ALGraph G, int i) {
    visit (i);                       // 访问初始顶点i
    visited[i] = TRUE;
    EnQueue (Q, i);                  // 顶点i入队
    while (!IsEmpty(Q)) {
        DeQueue (Q, v);              // 队首顶点v出队
        for (p = G.vertices[v].firstarc ; p ; p = p->nextarc) {
            w = p->adjvex ;          // 检测v的所有邻接点
            if (visited[w] == FALSE) {
                visit (w);           // w未访问过，标记，入队.
                visited[w] = TRUE;
                EnQueue (Q, w);
            }
        }
    }
}
```

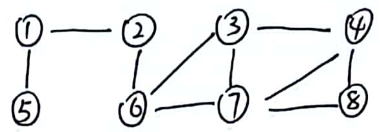
(3) 邻接矩阵实现 BFS： 时间复杂度 $O(|V|^2)$

```
void  BFS (MGraph G, int i) {
    visit(i);
    visited[i] = TRUE;
    EnQueue (Q, i);
    while (!IsEmpty(Q)) {
        DeQueue (Q, V);
        for (w=0; w<G.vexnum ; w++)
            if (visited[w] == FALSE && G.edge[v][w] == 1) {
                visit (w);
                visited[w] = TRUE;
                EnQueue (Q, w);
            }
    }
}
```

(4) 广度优先生成树：广度遍历过程中得到的遍历树。

① 同一个图的邻接矩阵存储表示是唯一的，故其广度优先生成树也是唯一的。

② 邻接表不唯一，～不唯一。



遍历过程：

~~step1~~：②

step2：② ① ⑥

step3：① ⑥ ⑤

step4：⑥ ⑤ ③ ⑦

step5：⑤ ③ ⑦

step6：③ ⑦ ④

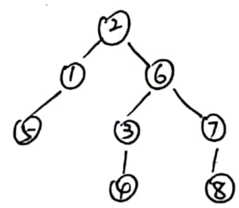
step7：⑦ ④ ⑧

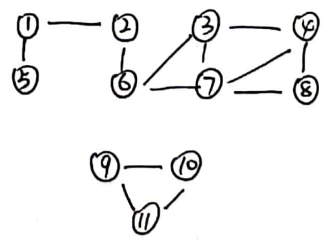
step8：④ ⑧

step9：⑧

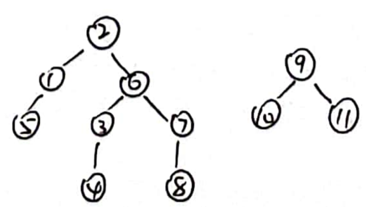
遍历序列：② ① ⑥ ⑤ ③ ⑦ ④ ⑧

生成树：





生成森林



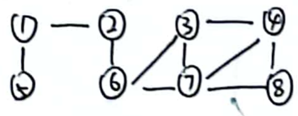
2. 深度优先搜索DFS

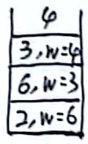
① 相当于树的先序遍历.

② 根结点是任意出发的结点.

③ 子结点是所有邻近的未访问过的结点.

④ 递归实现. 空间复杂度 $O(|V|)$





函数调用栈

从2开始的遍历序列:

②①⑤⑥③④⑦⑧.

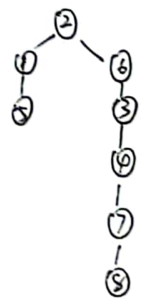
从1开始的遍历序列:

①②⑥③④⑦⑧⑤.

(1) DFS算法伪代码:

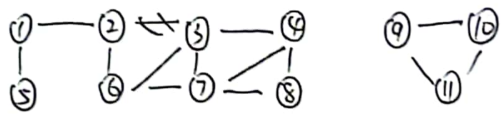
```
bool visited [MAX_VERTEX_NUM];    //访问标记数组
void DFSTraverse (Graph G){
    for(int i=0; i<G.vexnum; i++)
        visited[i] = FALSE;
    for (int i=0; i<G.vexnum; i++)   //从v0开始访问
        if (!visited [i])            //对尚未访问的顶点调用DFS
            DFS(G,i);
}
```

生成树(从2开始):



(2) 邻接表实现DFS: 时间复杂度 $O(|V|+|E|)$

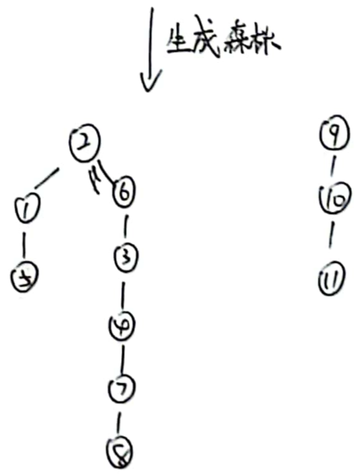
```
void DFS (ALGraph G, int i){
    visit (i);
    visited[i] = TRUE;                       //作访问标记
    for (p=G.vertices[i].firstarc; p; p=p->nextarc){
        j=p->adjvex;                         //检测i的所有邻接点
        if (visited[j]==FALSE)               //邻接点j未被访问过.
            DFS(G,j);                        //递归访问j
    }
}
```



(3) 邻接矩阵实现DFS: 时间复杂度 $O(|V|^2)$

```
void DFS (MGraph G, int i){
    visit (i);
    visited[i] = TRUE;
    for(j=0; j<G.vexnum; j++){
        if (visited[j]==FALSE && G.edge[i][j]==1)
            DFS (G,j);
    }
}
```

生成森林



(4) 深度优先生成树和生成森林

① 连通图 → 生成树.

② 非连通图 → 生成森林.

③ 基于邻接表存储的图, 生成树不唯一.

(BFSTraverse(), DFSTraverse()中)

[注]: BFS(G,i)或DFS(G,i)的次数等于该图的连通分量数 (无向图)

有向图的非强连通分量一次BFS(G,i)或DFS(G,i)无法访问到该连通分量的所有顶点.

四、图的应用

1. 最小生成树：带权连通无向图G权值之和最小的生成树。

① 最小生成树并不是唯一的(存在相同边时)

② 图G中各边权值不相等时，唯一。

③ 图G边数比顶点数少1，即G本身是一棵树时，本身就是最小生成树。

④ 权值之和唯一。

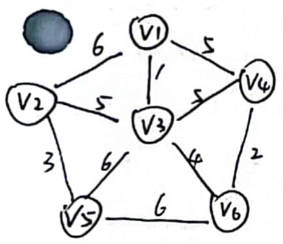
⑤ 最小生成树的边数 = 顶点数 - 1.

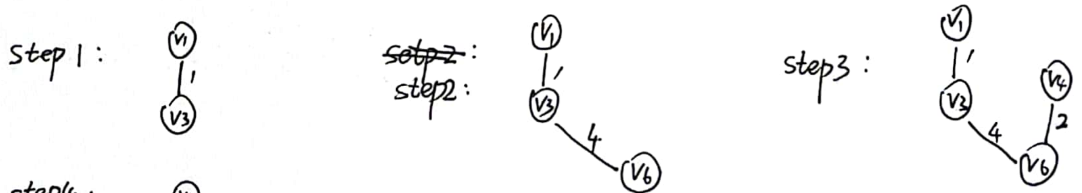
⑥ 不能保证任意两顶点之间的路径是最短路径。

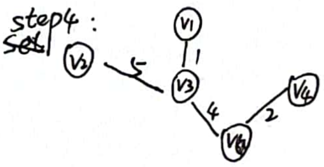
(1) Prim 算法：

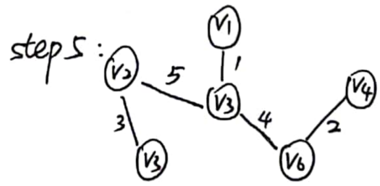
① 类似于寻找图的最短路径的Dijkstra算法(以自我为中心)。

② 当带权连通图的任意一个环中所包含的边权值不同时，最小生成树唯一。







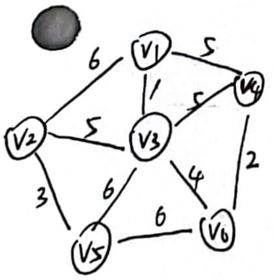


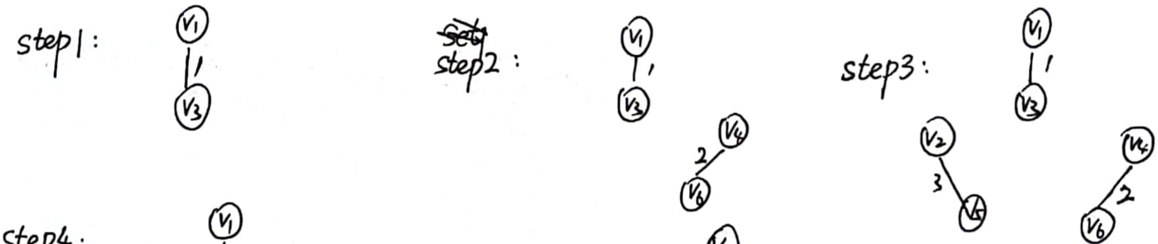
时间复杂度 $O(|V|^2)$

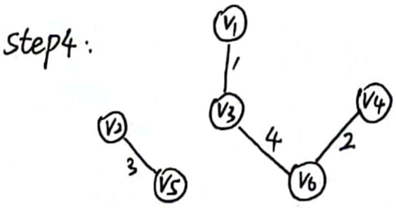
适用于求解边稠密图(从顶点开始)

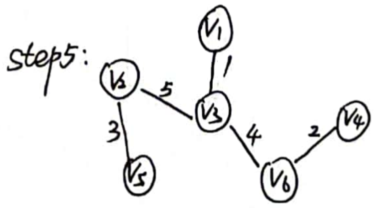
(2) kruskal 算法：

① 按权值的递增次序选择合适的边来构造最小生成树(不以自我为中心)









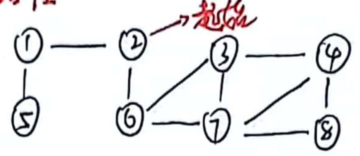
时间复杂度 $O(|E|\log_2|E|)$.

适用于求解边稀疏而顶点多的图(从边开始)

2. 最短路径

(1) BFS求无权图的单源最短路径问题

```
void BFS_MIN_Distance(Graph G, int u){
    for(i=0; i<G.vexnum; ++i)      // d[i]表示从u到i结点的最短路径
        d[i] = ∞;                  // 初始化
    visited[u] = TRUE; d[u]=0;
    EnQueue(Q, u);
    while(!IsEmpty(Q)){            // BFS算法过程
        DeQueue(Q, u);             // 队头元素u出队
        for(w=FirstNeighbor(G,u); w>=0; w=NextNeighbor(G,u,w))
            if(!visited[w]){       // w为u尚未访问的邻接顶点
                visited[w] = TRUE; // 标记
                d[w] = d[u]+1;     // 路径长度+1
                EnQueue(Q, w);     // w入队
            }
    }
}
```



	1	2	3	4	5	6	7	8
d[i]	1	0	2	3	2	1	2	3
path[i]	2	-1	6	3	1	2	6	7

path[i] → 记录前驱结点

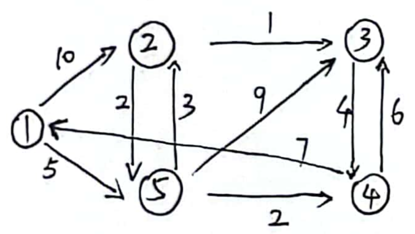
2到8: 8 <— 7 <— 6 <— 2 .

(2) Dijkstra求有向图的单源最短路径问题

① 基于贪心策略

② 时间复杂度 $O(|V|^2)$

③ 适用于带权有向、无向图，还适用于负权值图.



从V_1到各终点的dist值和最短路径的求解过程

顶点	第1轮	第2轮	第3轮	第4轮
2	10 $V_1 \to V_2$	8 $V_1 \to V_5 \to V_2$	8 $V_1 \to V_5 \to V_2$ ✓	
3	∞	14 $V_1 \to V_5 \to V_3$	13 $V_1 \to V_5 \to V_4 \to V_3$	9 $V_1 \to V_5 \to V_2 \to V_3$
4	∞	7 $V_1 \to V_5 \to V_4$ ✓		
5	5 $V_1 \to V_5$ ✓			
集合S	{1,5}	{1,5,4}	{1,5,4,2}	{1,5,4,2,3}

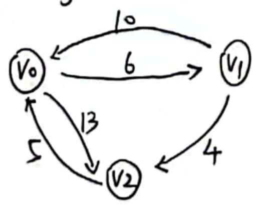
第1轮: 1->5, path=5

第2轮: 1->5->4, path=7

第3轮: 1->5->2, path=8

第4轮: 1->5->2->3, path=9

(3) Floyd求任意两顶点之间的最短路径问题.



有向图G

$$\begin{bmatrix} 0 & 6 & 13 \\ 10 & 0 & 4 \\ 5 & \infty & 0 \end{bmatrix}$$

G的邻接矩阵

$A^{(n-1)}$中保存了任意一对顶点之间的最短路径长度

A	$A^{(-1)}$			$A^{(0)}$			$A^{(1)}$			$A^{(2)}$		
	V_0	V_1	V_2	V_0	V_1	V_2	V_0	V_1	V_2	V_0	V_1	V_2
V_0	0	6	13	0	6	13	0	6	10	0	6	10
V_1	10	0	4	10	0	4	10	0	4	9	0	4
V_2	5	∞	0	5	11	0	5	11	0	5	11	0

核心算法过程: 行 中间顶点、 列

① if $A^{-1}[i][j] > A^{-1}[i][k] + A^{-1}[k][j]$

then $A^{-1}[i][j] = A^{-1}[i][k] + A^{-1}[k][j]$

② $A^{(0)}[i][j]$是从顶点V_i到V_j、中间顶点是V_0的最短路径的长度.

③ $A^{(k)}[i][j]$是从顶点V_i到V_j、中间顶点序号不大于k的最短路径长度.

④ 不断迭代的过程.

① 时间复杂度 $O(|V|^3)$

② 不能解决"负权回路"图

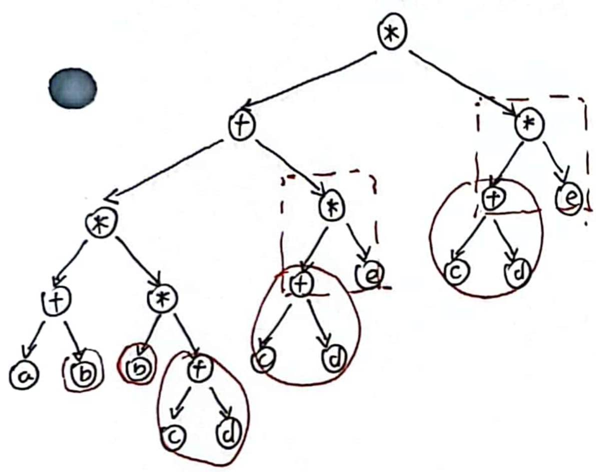
(3) 总结

	BFS	Dijkstra	Floyd
用途	求单源最短	求单源最短	求各顶点之间最短路径
无权图	✓	✓	✓
带权图	×	✓	✓
带负权值的图	×	×	✓
带负权回路的图	×	×	×
时间复杂度	$O(\|V\|^2)$ 或 $O(\|V\|+\|E\|)$	$O(\|V\|^2)$	$O(\|V\|^3)$

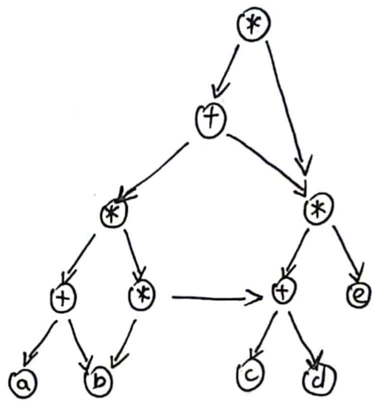
3. 有向无环图描述表达式

(1) 有向无环图定义：若一个有向图中不存在环，则称为有向无环图，简称DAG图。

(2) 利用有向无环图描述表达式： ((a+b)*(b*(c+d)) + (c+d)*e)*((c+d)*e)



二叉树描述表达式



有向无环图描述表达式

△不可能出现重复的操作数顶点.

4. 拓扑排序(优先入度为0)

(1) AOV网：若用有向无环图表示一个工程，其顶点表示活动，用有向边 <Vi, Vj> 表示活动 Vi 必须先于活动 Vj 进行的这样一种关系，则将这种有向图称为顶点表示活动的网络，简称AOV网。 Vi与Vj的直接前驱，前驱和后继关系具有传递性

(2) 拓扑排序：① 每个顶点出现且只出现一次。

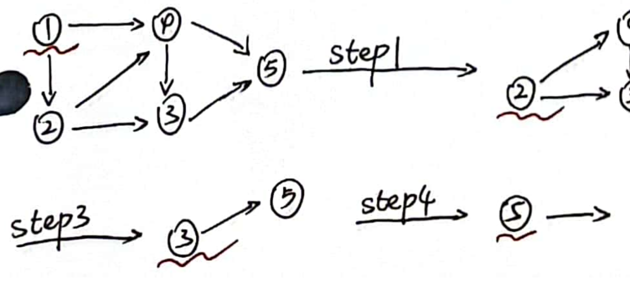
② 若顶点A在序列中排在顶点B的前面，则在图中不存在从B到A的路径。

③ 每个AOV网都有一个或多个拓扑排序序列。

时间复杂度 { 邻接矩阵：$O(|V|^2)$；邻接表：$O(|V|+|E|)$ }

(3) 核心算法：每轮选择一个入度为0的顶点并输出，然后删除该顶点和所有以它为起点的有向边。

可判断有向图中是否存在环.



最终拓扑排序 {1, 2, 4, 3, 5}

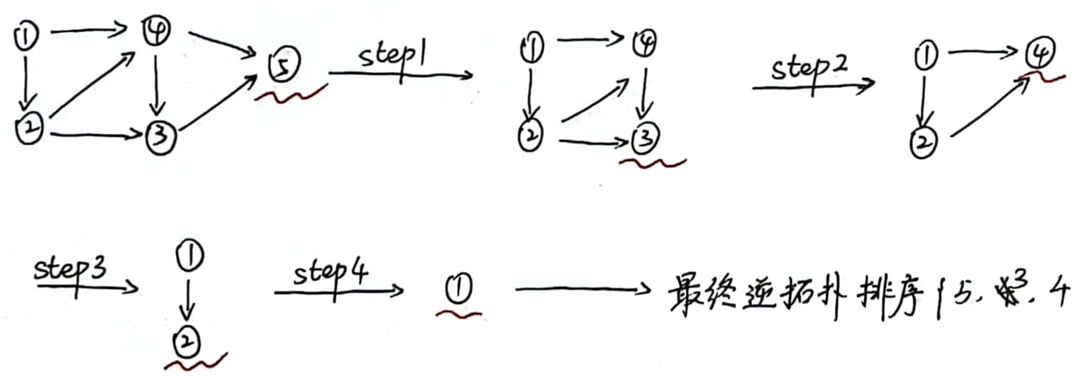
```
bool TopologicalSort (Graph G) {
    InitStack(S);      // 初始化栈，存入度为0的顶点
    int i;
    for (i=0; i<G.vexnum; i++) {
        if (indegree[i]==0)    Push(S,i);   // 将所有入度为0的顶点进栈
    }
    int count=0;      // 计数，记录当前已经输出的顶点数
    while (!IsEmpty(S)) {  // 存在入度为0的顶点
        Pop(S,i);     // 出栈
        print[count++]=i;  // 输出顶点i
        for (p=G.vertices[i].firstarc; p; p=p->nextarc) {
            v=p->adjvex;     // 将所有i指向的顶点的入度减1，并且将入度减为0的顶点压入栈S
            if (!(--indegree[v]))
                Push(S,v);     // 入度为0，则入栈
        }
    }
    if (count<G.vexnum)    return false;   // 排序失败，有向图中有回路
    else    return true;
}
```

5. 逆拓扑排序（优先出度为0）

核心算法：①从AOV网中选择一个出度为0的顶点并输出；

②从网中删除该顶点和所有以它为终点的有向边；

③重复①②，至AOV网空。



——→ 最终逆拓扑排序{5, 3, 4, 2, 1}.

[注]：①用DFS算法遍历一个无环有向图，并在退栈返回输出相应的顶点，也可得到逆拓扑排序序列。

②若一个顶点有多个直接后继，则拓扑排序结果通常不唯一。

③一个有向图具有有序的拓扑排序序列，则它的邻接矩阵为三角矩阵。

6. 关键路径

(1) AOE网：在带权有向图中，以顶点表示事件，以有向边表示活动，以边上的权值表示完成该活动的开销（如完成活动所需的时间），称之为用边表示活动的网络，简称AOE网。

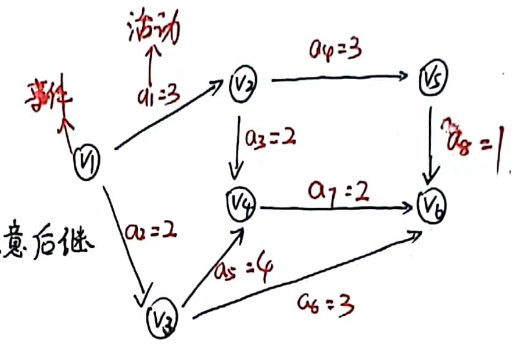
① 仅有一个入度为0的顶点，称为开始顶点（源点）

② 仅有一个出度为0的顶点，称为结束顶点（汇点）。

从源点到汇点的所有路径中，具有最大路径长度的路径称为关键路径，而把关键路径上的活动称为关键活动。

长度是完成整个工程的最短时间

(2) 相关参量



① 事件 v_k 的最早发生时间 $ve(k)$：

指从源点 v_1 到顶点 v_k 的最长路径长度。

$$\begin{cases} ve(\text{源点})=0 \\ ve(k)=\max\{ve(j)+\text{Weight}(v_j,v_k)\} \end{cases}$$ ，v_k 为 v_j 任意后继

$ve(1)=0$　$ve(3)=2$　$ve(2)=3$　$ve(5)=6$

$ve(4)=\max\{ve(2)+2,\ ve(3)+4\}=6$

$ve(6)=\max\{ve(5)+1,\ ve(4)+2,\ ve(3)+3\}=8$.

拓扑排序：$v_1, v_3, v_2, v_5, v_4, v_6$

逆拓扑排序：$v_6, v_5, v_4, v_2, v_3, v_1$

② 事件 v_k 的最迟发生时间 $vl(k)$：

$$\begin{cases} vl(\text{汇点})=ve(\text{汇点}) \\ vl(k)=\min\{vl(j)-\text{Weight}(v_k,v_j)\} \end{cases}$$ ，v_k 为 v_j 任意前驱

$vl(6)=8$　$vl(5)=7$　$vl(4)=6$

$vl(2)=\min\{vl(5)-3,\ vl(4)-2\}=4$

$vl(3)=\min\{vl(4)-4,\ vl(6)-3\}=2$

$vl(1)=0$

	v_1	v_2	v_3	v_4	v_5	v_6
$ve(i)$	0	3	2	6	6	8
$vl(i)$	0	4	2	6	7	8

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
$e(i)$	0	0	3	3	2	2	6	6
$l(i)$	1	0	4	4	2	5	6	7
$l(i)-e(i)$	1	0	1	1	0	3	0	1

根据 $l(i)-e(i)=0$ 的关键活动，得关键路径 (v_1, v_3, v_4, v_6)

③ 活动 a_i 的最早开始时间 $e(i)$：

指该活动弧的起点所表示的事件的最早发生时间，若 $\langle v_k, v_j\rangle$ 表示活动 a_i，则 $e(i)=ve(k)$.

④ 活动 a_i 的最迟开始时间 $l(i)$：

指该活动弧的终点所表示的事件的最迟发生时间与该活动所需时间之差。

若 $\langle v_k, v_j\rangle$ 表示活动 a_i，则 $l(i)=vl(j)-\text{Weight}(v_k,v_j)$.

⑤ 活动 a_i 完成的时间余量 $d(i)=l(i)-e(i)$